Introductory Robot Programming

Final Project Report - GROUP 8

Path Planning for Robot in a Maze using Breadth First Search

Akash Agarwal (116904636) Jevay Aggarwal (116788953)

Preetham Patlolla (116197571) Sanket Acharya (116265515)

Smriti Gupta (116748612) Varsha Eranki (116204569)

Contents

List of Figures

1 Introduction

- A robot is navigated through a maze under dynamic obstacles from a start node to a goal node fed by the user,(which is usually directed towards the center of the maze as its goal node).
- The Micromouse Simulator is implemented for the GUI interpretation of the path in the maze. It is used for navigating the two types of mobile robots comprising of Wheeled and Tracked attributes.
- The program exits when no path can be formed between the current position of the robot to its goal node or when it reaches the goal node.
- We denote the current/start position of the robot with "S" and the goal position is denoted with "G".
- The prominent Object oriented programming concepts of Inheritance, Polymorphism, Abstraction and Encapsulation is implemented by the creating classes to store the data privately, and access the data in derived classes as per the requirement. Polmorphism implementation is seen in virtual function methods and also in pointers and reference objects to base class.
- The code is generated using Breadth First Search(BFS) as a search algorithm for the robot navigation to the goal node in the maze.
- An API class is created which consists a set of functions and procedures allowing the creation of applications that access the features or data of an operating system, application, or other services.

2 Overview of the approach

The project has a main source file and a source directory with the defined five source files and six header files as stated below:

- **API, Algorithm, LandBasedTracked, LandBasedWheeled and Maze** are the source files for the method implementations of the method definitions defined in the header files.
- **Maze, LandBasedWheeled, LandBasedRobot, LandBasedTracked, Algorithm, API** are the header files where the method prototypes are defined.

LandBasedRobot.h header file has the LandBasedRobot abstract class defined in it and it uses namespace as *"fp"* which means the Final Project. Common attributes and methods for its derived classes are defined, deep copy constructor is used to initialize followed by its destructor. Method prototypes for class are also defined as virtual functions and are overridden in their derived class header files.
Moreover, Accessors and Mutators are defined for the attributes and methods respectively to be able to access and change the data.

LandBasedTracked.h and **LandBasedWheeled.h** are the hader files with their concrete derived classes of the same name from the above base class. Their respective constructors and destructors are again called for the newly defined attributes and methods in the derived classes, along with their Acecessors and Mutators.

The **API.h** header file in namespace *"fp"* which has the class API where the dimensions of the maze from the micromouse simulator, obstacle placing, robot navigation based on the obstacle interaction and obstacle clearance are defined. It is a class that basically bridges the code and the micromouse simulator.

The **Maze.h** heaader file has a 2-dimensional array defined as $256X4$ to store the structure of the micromouse simulator. The first element of the array stores the coordinates of the maze, while the next element of the array stores the direction of the obstacle corresponding to $0, 1, 2 and 3$ for *"North, East, South and West"* directions respectively. It also has the coordinates of the Robot as well as the accessors for the maze dimensions defined.
Apart from these, we have method prototypes for getting the coordinates and setting the coordinates for the dynamic obstacles in order to calculate the robot path in the Algorithm class.

The **Algorithm.h** header file defines our class that will define and implemement the Breadth First search(BFS) algorithm in to traverse through the maze under dynamic obstacles. It has the determined path array defined in its attributes. Its methods comprise of Path checking, generate the path after checking the conditions for obstacles and moving the robot after these operations.

The project is designed based on OOP implemenations. Its has six classes stated as **LandBasedRobot, LandBasedWheeled, LandbasedTracked, API, Maze and Algorithm.**

2.1 Classes Implemented

All the attributes of the base and derived classes are declared in protected format, and all methods of the base and derived classes are declared in public format. The following classes were created to implement the program:

2.1.1 LandBasedRobot

- It is a abstract class with its derived classes as *LandBasedWheeled* and *LandBasedTracked*. Its attributes and methods are defined in protected and public format respectively to be implemented in its derived classes.

- The attributes in the class are, *"name_ , speed_ , width_ , length_ , height_ , capacity_ , x_ , y_ , direction_."*
- The *"name_"* attribute takes a string datatype defining the name of the robot.
- The *"speed_ , width_ , length_ , height_ , capacity_"* attributes have the double datatype that input the driving speed, base width, base length, base height of the robot and payload of the arm, respectively.
- *"x_, y_"* integer datatype attributes give the coordinates of the robot in the maze.
- The *"direction_"* is a character datatype attribute that gives the direction in which the robot is facing in the maze. The directions are denoted by *"N","E","W","S"* for North, East, West, South directions respectively.
- The methods in the class are, *"GetDirection(), MoveForward(), TurnLeft(), TurnRight()."* They have identifiers that state their corresponding function straightforwardly for the robot navigation.

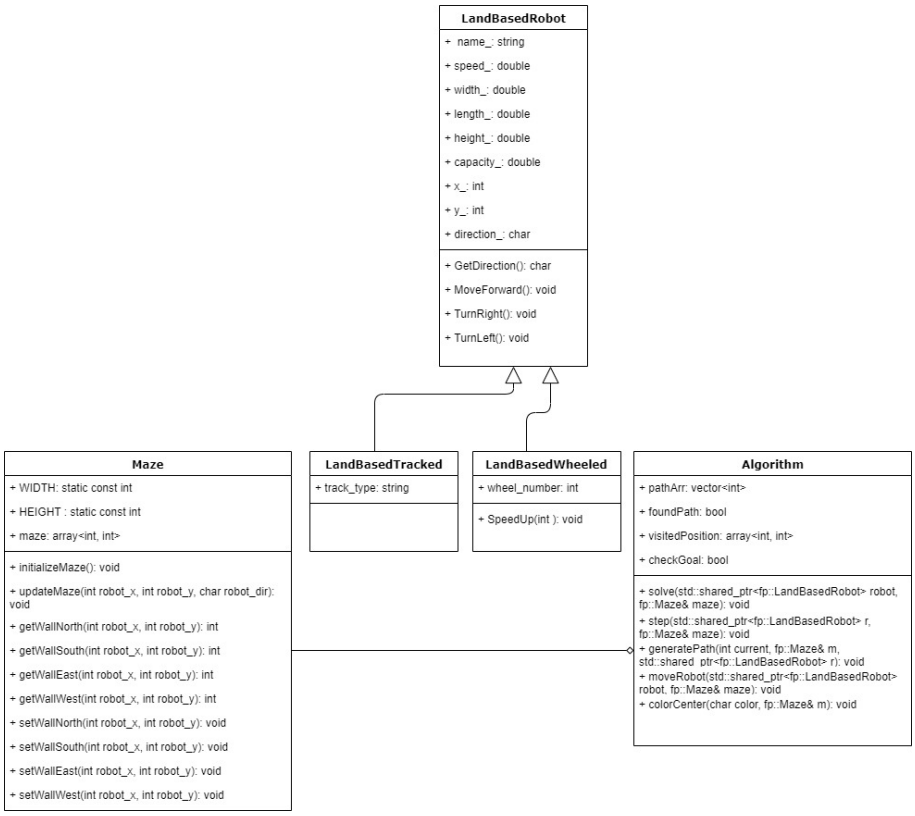

Figure 1: Classes Implemented - Attributes and Methods

2.1.2 LandBasedWheeled

- This concrete class is derived from the base class *LandBasedRobot* and inherits its attributes and methods.
- Another integer datatype attribute is defined in this class, *"wheel_number"*, that counts the number of wheels on the robot.
- A method is defined in the class as well, *"SpeedUp(int)"*, which allows the robot to increase its speed by travelling more number of cells in each step. It has no return type(hence, void) and takes in an integer argument as speed from its attribute.

2.1.3 LandBasedTracked

- This concrete class is derived from the base class *LandBasedRobot* and inherits its attributes and methods.
- An additional string datatype attribute is defined, *"track_type"*, that specifies the type of track mounted on the robot.

2.1.4 Maze

- After reading the maze loaded in the Micromouse Simulator, this class stores the maze structure in the form of array.
- The method source file consists of initializing the coordinates of the maze, setting its coordinates, and checking for obstacles in the maze to navigate the robot through a obstacle free path.
- These operations are carries out for all four directions of West(*"W"*), South(*"S"*), North(*"N"*) and East(*"E"*).
- When the robot encounters obstacles(*"walls"*), its respective final project namespace is called in the API source file and the direction is returned for the robot navigation to happen.

2.1.5 API

- API class defines the dimensions of the Maze, dynamic obstacle placing, robot's direction for navigation and also the color settings for the navigated path explored and the dynamic obstacles in maze.
- Apart from these, we also define a reset button to get back to the original state of the robot in the maze.

2.1.6 Algorithm

- Algorithm class is a derived class from all the other four classes. It stores the array indexes of the robot path in the form of an array.
- It takes the robot positions at every iteration in the maze, and checks if the current position is the goal position, and runs the program in the direction preference as mentioned ("n", "e", "s", "w") till it matches. It exits when the robot path cannot be determined or when it reaches the goal.
- For path generation, the Manhattan distance to the goal node is calculated to optimize the robot in navigating its path towards the goal.
- For bactracking, we add the goal that was found to the path and push back to the initial node. The robot is moved based on the current direction, the direction of the next cell relative to the current cell, if the obstacle is encountered there or at the end goal.
- We store the obstacle space in a two dimensional array format of size 256x4 where 256 sized array represents the coordinates of the maze and the 4 arrays represent the four directions of the obstacles possible for the robot. 1 represents that there is an obstacle in its respective directioned array and 0 represents the absence of the obstacle.
- With the help of this, the robot then makes a decision to take a step as per the priority listed. The path and the obstacle walls are uniquely color coded for easy diffrentiation.
- Backtracking is done by tracing back(reversing) to the visited nodes with their parent nodes to reach the start node.

2.2 Flowchart of the Project

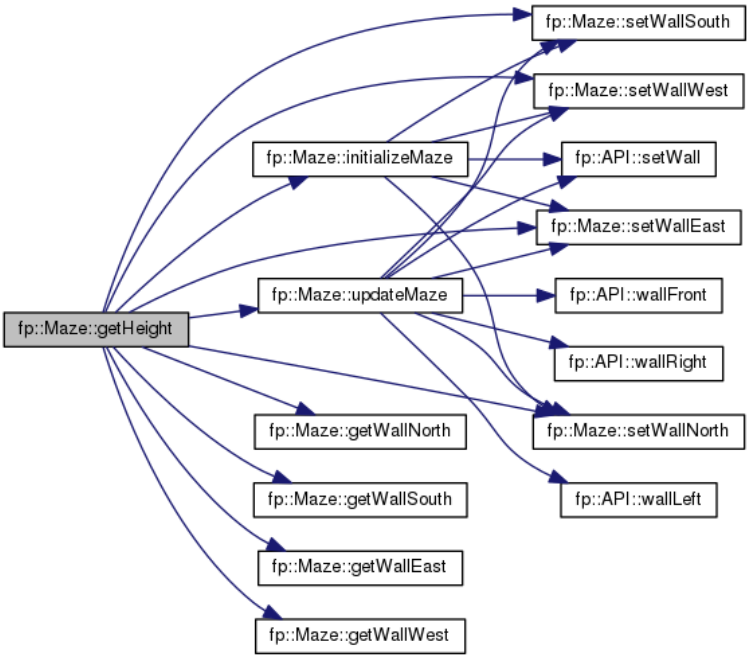

Figure 2: Member Functions in Maze.h - For updations in Micromouse Simulator

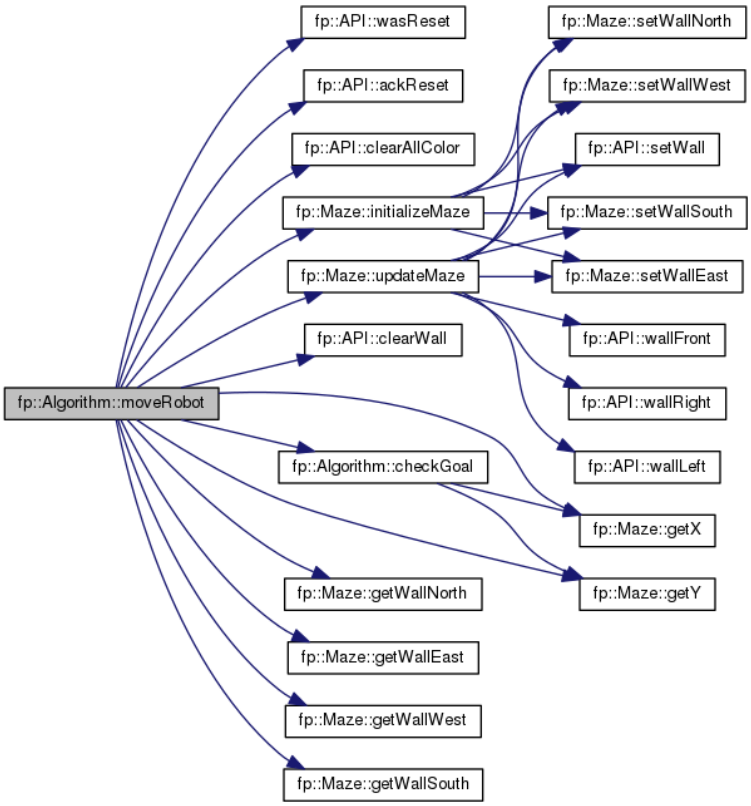

Figure 3: Member Functions in Algorithm.h - Defining BFS

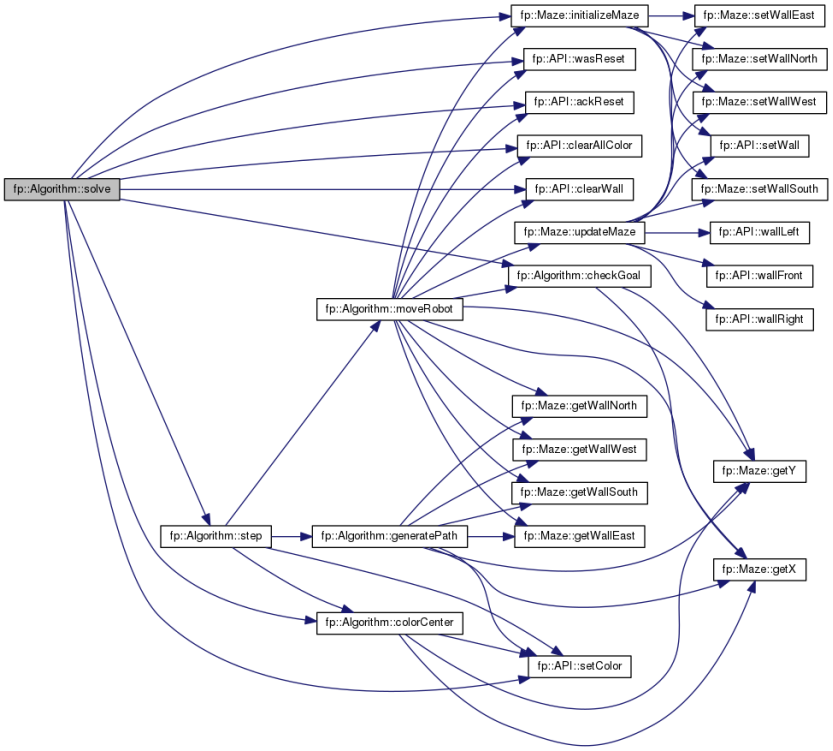

Figure 4: Member Functions in Algorithm.cpp - Flow of Algorithm

2.3 Breadth First Search-Pseudo Code

2.3.1 Pseudo Code of the BFS Algorithm implemented

initializing the start and goal nodes;

checking if the current position of the robot is the goal;

for BFS(G,start_v):

say Q be a queue;

label start_v as discovered;

Q.enqueue(start_v);

while Q is **not empty**:

v = Q.dequeue();

if v is the goal:

return v;

for all edges from v to w in G.adjacentEdges(v) **do**:

if w is not labeled as discovered:

label w as discovered;

w.parent = v;

Q.enqueue(w);

2.3.2 Pseudo Code of the project

initialize the maze (set walls around the perimeter);
color the center of the maze;
initialize the robot (position and direction);
while *true* **do**
    clear all tile color;
    read all the walls;
    get the current cell of the robot;
    generate a path from current cell to destination (BFS);
    if *no path from current position to destination* **then**
        unsolvable maze;
        exit;
    end
    draw path in the maze with **API::setColor(int,int,char);**
    move the robot along the path with *TurnLeft(), TurnRight() and MoveForward()*;
    if *robot reaches destination* **then**
        success;
        exit;
    else
        set new walls;
    end
end

We chose Breadth First Search as it explores all of the neighbor nodes at the present depth prior to moving on to the nodes at the next depth level. It also consumes lesser time than Depth First search. .

3 Micromouse Simulator

We communicate with the Micromouse Simulator in the abstarct and derived classes of LandBasedRobot, LandBasedWheeled, LandBasedTracked, and API and Maze where the robot position is generated along with the obstacle placing is being called.

4 Contributions

4.1 Akash Agarwal

- Contributed towards the source code for LandBasedWheeled, LandBasedTracked. Helped on the project as well.

4.2 Jevay Aggarwal

- Contributed towards the source code for Algorithm and Maze files. Also generated the Doxygen file and Presentation.

4.3 Preetham Patlolla

- Contributed towards the source code for Algorithm and Maze files. Also created the Presentation.

4.4 Sanket Acharya

- Contributed towards the source code for Algorithm and Maze files. Also worked on the flowchart for the classes on the report.

4.5 Smriti Gupta

- Contributed towards the source code for LandBasedWheeled, LandBasedTracked. Helped with the presentation as well.

4.6 Varsha Eranki

- Contributed towards the source code for LandBasedWheeled, LandBasedTracked and Maze. Also worked on the Project report and Doxygen.

5 Discussions

There were some challenges faced during the project, they are discussed below:

1. Starting off with the byte maze was complicated as we had to consider too many parameters for path generation.
2. Absence of a debugger in the Micromouse Simulator led to difficulty in debugging and checking errors.
3. Initially we were not sure on how to dynamically choose one of the goal positions. Later, we decided to qualify one of the goal positions based on the Manhattan distance from the then start node.

4. The robot, even after reaching one of the goal positions, tended to move a step further. This was because in moving the robot forward we didn't check if the position it is coming from was already a goal.

5. Because of improper initialization of maze type std::array, there were junk values at wall positions leading to unexpected crashes while moving robot in maze.

6 Future Improvements

6.1 Overtime Modifications

- If there was an availability of resources, We would have implemented a better search algorithm for optimality like A*, Dijkstra and D*.

- Also, if time and resources permit, an improvement to look forward to was to implement the project in ROS Gazebo and tweak the SpeedUp parameter in the virtual environment under dynamic obstacles.

6.2 Implementations in Micromouse Simulator

- A debugger should have been a valuable feature for Micromouse Simulator.

7 Feedback on 809Y-Introductory Robot Programming

This coursework has been greatly helpful in bolstering our C++ fundamentals. It was mainly a great practice for us to work on Object oriented concepts which is a highly demanded prerequisite for those of us who are inclined to work in the software development field of Robotics.

Although, most of our teammates have some experience with C and the very basics of C++, and through this course we encountered tools like pointers, virtual functions, objects, classes, constructors and destructors which are the fundamentals of Object oriented programming. Not only did we learn them, but also had an experience to practise them in our last two projects. Coming to the aspects that we suggest require improvements are:

- More time allocation for Object oriented programming concepts and less time on functions, datatypes and structures.

- An introduction to stub implementations in C++ and more programmed coursework for pointers and references as these topics are bound to confuse many of us.

Concluding this project and getting towards the end of the course, We are grateful to the Prof. Zeid Kootbally and our Teaching assistants Utsav Patel and Abhishek Kathpal for their timely help and guidance. Group 8 looks forward to exploring and learning the other deeper concepts of C++ and ROS in the coming future.